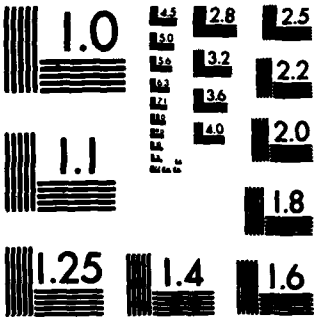
AD-A144 360 PROGRAM VERIFICATION(U) TEXAS UNIV AT AUSTIN INST FOR 1/1
COMPUTING SCIENCE AND COMPUTER APPLICATIONS
R S BOYER ET AL. 1984 N00014-81-K-0634

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD-A144 360

DTIC FILE COPY

PROGRAM VERIFICATION

Robert S. Boyer and J Strother Moore

To Appear in the Journal of Automated Reasoning

DTIC ELECTE AUG 15 1984 S D A

This document has been approved for public release and sale; its distribution is unlimited.

The research reported here was supported by National Science Foundation Grant MCS-8202943 and Office of Naval Research Contract N00014-81-K-0634.

Institute for Computing Science and Computer Applications
The University of Texas at Austin
Austin, Texas 78712

84 07 24 012

Table of Contents

Computer programs may be regarded as formal mathematical objects whose properties are subject to mathematical proof. Program verification is the use of formal, mathematical techniques to debug software and software specifications.

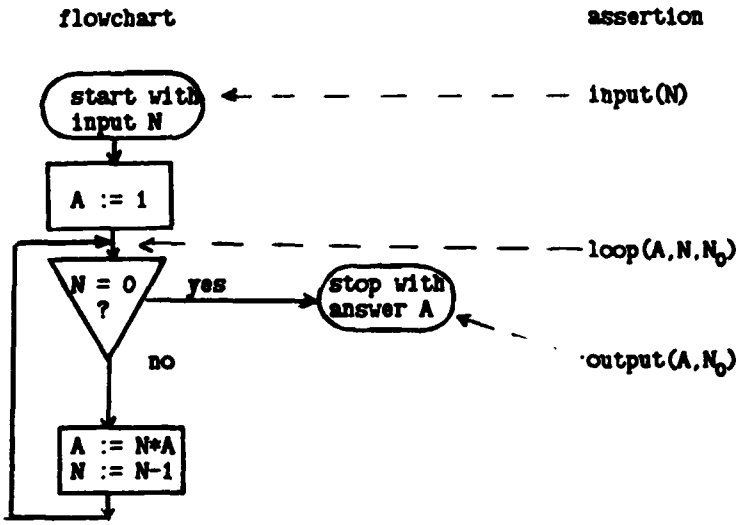
1. Code Verification

How are the properties of computer programs proved? We discuss three approaches in this article: inductive invariants, functional semantics, and explicit semantics. Because the first approach has received by far the most attention, it has produced the most impressive results to date. However, the field is now moving away from the inductive invariant approach.

1.1. Inductive Assertions

The so-called Floyd-Hoare inductive assertion method of program verification [25, 33] has its roots in the classic Goldstine and von Neumann reports [53] and handles the usual kind of programming language, of which FORTRAN is perhaps the best example. In this style of verification, the specifier "annotates" certain points in the program with mathematical assertions that are supposed to describe relations that hold between the program variables and the initial input values each time "control" reaches the annotated point. Among these assertions are some that characterize acceptable input and the desired output. By exploring all possible paths from one assertion to the next and analyzing the effects of intervening program statements it is possible to reduce the correctness of the program to the problem of proving certain derived formulas called verification conditions.

Below we illustrate the idea with a simple program for computing the factorial of its integer input N



Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A1
DSIS COPY INSPECTED

The first assertion is the "input assertion" and might be something like $0 \leq N$. When proving this program correct the input assertion may be assumed for the initial value of N, N_0. The second assertion is the "loop invariant" and states the relations that hold among the variables. The third assertion is the "output assertion;" in this program it is $A = N_0!$, where the mathematical definition of N! is provided axiomatically:

$$N! = \begin{cases} 1, & \text{if } N = 0 \end{cases}$$

```
N! = <
     |  N*(N-1)!, otherwise
     \
```

The verification conditions for this problem are shown below.

```
path from input to loop:
input(N0) -> loop(1,N0,N0)

path from loop to loop:
loop(A,N,N0) & N≠0 -> loop(N*A,N-1,N0)

path from loop to output:
loop(A,N,N0) & N=0 -> output(A,N0)
```

It is claimed that if these three formulas are theorems then whenever the program is started on a nonnegative integer N, the final value of A is N!, provided the program terminates.

The transformation of a properly annotated flowchart into verification conditions can be done mechanically. Such a program is called a verification condition generator. The flowchart is usually presented as a program in some ordinary programming language. Mechanical verification systems based on the inductive assertion method usually consist of two main subsystems: a verification condition generator and an automatic theorem-prover or proof checker to prove the verification conditions.

The first mechanical program verification system was developed by King [36], a student of Floyd's. Many verification systems have been developed since [28, 35, 30, 20, 6].

Using techniques similar to the generation of verification conditions it is possible to prove termination and absence of runtime errors. Consider for example the claim made for the system described in [6]:

> *If a FORTRAN subprogram is accepted and proved by the system and the program can be loaded onto a FORTRAN processor that meets the ANSI specification of FORTRAN [52, 1] and certain parameterized constraints on the accuracy of arithmetic, then any invocation of the program in an environment satisfying the input condition of the program will terminate without run-time errors and will produce an environment satisfying the output condition of the program.*

The verification conditions generated are proved by the Boyer-Moore theorem prover [5]. Among the FORTRAN programs proved correct mechanically by the above described system are a fast string searching algorithm [6], an integer square root algorithm using Newton's method [7], and a linear time majority vote algorithm [9]. These programs are each relatively small, requiring no more than a page of code. However, the correctness arguments are fairly deep.

Two of the most widely known verification systems, the Stanford Verifier by David Luckham and his students at Stanford University and the Gypsy Verification Environment by Don Good and his colleagues at the University of Texas at Austin, have been used to verify significantly larger programs. Unlike the FORTRAN *verifier* above, these two *verification systems* present the user with an integrated set of tools and specially tailored programming languages designed to make verification more convenient.

The programming language supported by the Stanford Verifier [35] is a variant of PASCAL. The theorem-prover used is a rewrite rule based simplifier built on a decision procedure by Oppen and Nelson [44]. The most significant verification task accomplished with that system to date is the verification of a compiler for PASCAL, by Polak [47]. The total amount of executable code verified in that application is around 3000 lines.

The Gypsy Verification Environment (GVE) supports the programming language Gypsy, which is a derivative of Pascal providing a somewhat cleaner semantics and concurrency [29]. The theorem-prover used in GVE was adapted from a prover by Bledsoe [4]. GVE was used to verify the largest program mechanically verified to date: a communications interface to a computer network [48]. The interface consists of over 4200 lines of executable code. Some 2600 verification conditions were proved mechanically in that effort. GVE has also been used to verify a "message flow modulator" which, in simple terms, is a switch on a communications line that shunts to a branch line messages with certain properties.

1.2. Functional Semantics

Another approach to program verification is to transform the flowchart into a mathematical function from the input state to the output state. This approach was initially suggested by McCarthy [40].

The loop in the flowchart presented above can be transformed into the following recursive function provided the input is known to be a natural number and one assumes that the value of A is the only interesting component of the final state:

$$\text{loop}(N,A) = \begin{cases} A, & \text{if } N=0 \\ \text{loop}(N-1, N*A), & \text{otherwise} \end{cases}$$

Since the loop of the flowchart is entered after initializing A to 1, the expression loop(N,1) represents the final value of A computed by the program.

This transformation process can be carried out mechanically by a program that, like a verification condition generator, knows the semantics of the programming language. Once a program has been transformed into a function its properties may be proved by standard mathematical techniques. For example, the specification of the factorial program is loop(N,1) = N!.

This theorem is easily proved by first proving, by induction, the more general theorem loop(N,A) = A*N!.

While this approach can in principle be applied to any programming language with a well understood semantics, it is in fact most often applied to applicative programming languages where the initial step -- the transformation of a program into a function -- is unnecessary.

The Boyer-Moore theorem-prover has been used to prove properties of many recursive functions. Among the theorems proved are the invertibility of the Rivest, Shamir, and Adleman public key encryption algorithm [11], the soundness and completeness of a propositional calculus decision procedure [5], the soundness of an arithmetic simplifier now used in the system [8], the termination of Takeuchi's function [42], and the correctness of many elementary list and tree processing functions.

Examples of other work in this area include [15, 17, 2]. The Edinburgh LCF system (discussed below) has also been used to prove properties of recursive functions; among the theorems proved with LCF are the correctness of a parser and the correctness of the unification algorithm [19, 46].

1.3. Explicit Semantics

In both the inductive assertion and functional program verification methods, the semantics of a von Neumann language are embedded in some algorithm such as a verification condition generator. An alternative approach is to make programs be objects in the logic (typically tree structures) and then to spell out the semantics of programs explicitly with axioms. Scott-Strachey denotational semantics [50] is an example of this approach to program verification. For example, the Scott-Strachey semantics of a programming language containing an assignment statement [var := expr] is likely to have an axiom such as:

```
eval([var := expr;s],a) = eval(s,bind(var,val(expr,a),a))
```

which asserts that the meaning, in environment a, of a sequence of statements beginning with the assignment [var := expr], is the meaning of the tail of the sequence in the environment obtained from a by assigning the variable var the value of expr in a. In this setting, a program verification system consists simply of a mechanized logic together with the axioms for defining semantics.

The Edinburgh LCF system [31] is a mechanization of Scott's Logic for Computable Functions in which one can define the semantics of programming languages. The LCF system is similar to a proof checker in that it provides low level primitives for manipulating formulas in the logic, rather than a sophisticated heuristic search strategy. However, LCF also provides an extremely flexible metalanguage in which one can build (i.e., program) proof procedures or theorem-provers. The Edinburgh LCF has been used to prove the equivalence of two different semantic definitions [18].

Often the explicit definition of the semantics takes the form of an interpreter for the programming language. For example, in [12, 10] Boyer and Moore define an interpreter for Pure LISP and then use their mechanical theorem-prover to prove that Pure LISP is Turing complete and that the halting problem for Pure LISP is unsolvable.

2. Other Program Verification Applications

Thus far we have concentrated on the verification of sequential programs with respect to some specifications. There are several other related areas of program verification that we will simply mention here.

- The mechanical verification of the properties of abstract data types. See [16, 43, 26, 27].
- The mechanical verification of concurrent processes or networks of programs. Among the seminal papers devoted to theoretical understanding of concurrency are those by Owicki and Gries [45], Lamport [38], and Hoare [34]. Mechanical proofs of properties of concurrent systems, in addition to the previously cited network interface proof by Good with the Gypsy system, include the verification of protocols by Divito [23] and the proof of the correctness of a concurrent sort routine by Lengauer [39].
- The mechanical verification of properties of specifications. Since specifications are often simpler to reason about than programs, there have been several attempts to reason mechanically about specifications. This method has been used to try to establish the "security" of operating system designs. One such checker is that by Feiertag [24]. The idea of "design verification" was also used in the attempt to establish the reliability of SRI's Software Implemented Fault Tolerant (SIFT) system [49]. Of course, a program whose design has been verified is unworthy of trust until the running program has been shown to implement the design. Especially to be distrusted are those software products constructed by two unrelated teams: those who write the code and those who simultaneously and independently write the formal specifications, which are checked for security. Alas, several such projects are currently

funded by the U.S. government. This travesty of mathematical proof has been defended by a noted logician as at least giving the government better documentation. The Department of Defense has published official standards authorizing this nonsense.

3. Problems and Current Directions

Several important programming areas have been virtually ignored in program verification. One such area is the the mechanical verification of real-time control programs. A minor investigation into the area was done by Boyer and Moore in [13] in which they used their theorem-prover to prove that a simple program keeps a vehicle "on course" in a varying cross wind. A major problem in real-time control verification is the specification of the non-digital world with which such programs interact. A related area of concern is hardware verification, where timing and interrupt handling are major problems. An initial investigation was conducted by Wagner [54], who used Weyhrauch's FOL system [56] to prove properties of circuits for such basic tasks as counting and multiplying. Finally, an extremely important area that has received almost no attention is the mechanical verification of floating point algorithms.

Despite the successes of program verification, there is a widespread feeling that the use of verification conditions and the use of today's complicated von Neumann programming languages are major impediments to progress. The inductive assertion method suffers the disadvantage that the user's input -- a program and its specification -- is transformed into a mass of formulas which bear little resemblance to either. When one of these formulas fails to be a theorem the user must deduce whether the bug is in the program or the specification and what to do about it. This is a task similar to deducing the location of a black body by observing the effect of its gravitational pull on visible neighbors. The semantics of today's programming languages further complicates the problem.

The adoption of "applicative" or "functional" programming languages eliminates the first problem because proofs can be conducted in the language in which the program is written. In addition, applicative languages tend to be simpler than von Neumann languages. The simplicity of program verification in the setting of an applicative programming language has contributed to the growing interest in such languages. Among the applicative languages currently being developed are SASL by Turner [51], the reduction language of Backus [3], LISPKIT by Henderson [32], and the many variants of predicate calculus as a programming language [37]. However, applicative languages are widely considered to be too inefficient for many applications and much of the work in program verification at the moment is actually addressed at improving their efficiency. One of the methods used is to exploit their simple semantics and use automated reasoning to deduce the correctness of optimizations [21, 41].

A question that frequently comes up is "Have you verified the verifier?" Perhaps surprisingly, this metamathematical question is often asked by engineers (e.g., project monitors in NASA and the FAA) not merely by pointy headed academics. Of course, if a machine ever answers the question "Do you ever lie?" the answer will be no more informative than when a human answers the question. However, several approaches to "verified verifiers" are being pursued, ranging from running the output of the theorem-prover through a primitive but trusted proof checker to bootstrapping from such a proof checker to a verified automatic theorem prover [22, 55, 14, 8, 31].

The difficulties of verification notwithstanding, there is widespread interest in the field. The U.S. Government has already issued R.F.Q.'s requiring various forms of mechanical verification. In addition, the recently established Department of Defense Computer Security Center has defined several levels of software certification, the highest level being mechanical code verification. The potential market for practical program verification technology can be glimpsed by considering the number of everyday products, ranging from ballpoint pens to automobiles, containing microchips and then considering the cost

of recalling those products due to bugs in their software.

References

1. American National Standards Institute, Inc. American National Standard Programming Language FORTRAN. Tech. Rept. ANSI X3.9-1978, American National Standards Institute, Inc., 1430 Broadway, N.Y. 10018, April, 1978.

2. R. Aubin. *Mechanizing Structural Induction.* Ph.D. Th., University of Edinburgh, 1976.

3. J. Backus. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." *Comm. ACM 21* (August 1978), 616-641.

4. W.W. Bledsoe, P. Bruell. A Man-Machine Theorem-Proving System. Advance Papers of Third International Joint Conference on Artificial Intelligence, 5-1 (Spring), 1974.

5. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

6. R. S. Boyer and J S. Moore. A Verification Condition Generator for FORTRAN. In *The Correctness Problem in Computer Science,* R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

7. R. S. Boyer and J S. Moore. The Mechanical Verification of a FORTRAN Square Root Program. SRI International, 1981.

8. R. S. Boyer and J S. Moore. Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures. In *The Correctness Problem in Computer Science,* R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

9. R. S. Boyer and J S. Moore. MJRTY - A Fast Majority Vote Algorithm. Technical Report ICSCA-CMP-32, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1982.

10. R. S. Boyer and J S. Moore. A Mechanical Proof of the Unsolvability of the Halting Problem. Technical Report ICSCA-CMP-28, University of Texas at Austin, 1982. To appear in the *Journal of the Association for Computing Machinery.*

11. R. S. Boyer and J S. Moore. "Proof Checking the RSA Public Key Encryption Algorithm." *American Mathematical Monthly 91,* 3 (1984), 181-189.

12. R. S. Boyer and J S. Moore. A Mechanical Proof of the Turing Completeness of Pure Lisp. Technical Report ICSCA-CMP-37, Institute for Computing Science and Computer Appplications, University of Texas at Austin, 1983. To appear in the *Automated Theorem Proving* volume of the Contemporary Mathematics Series of the American Mathematical Society.

13. R. S. Boyer, M. W. Green and J S. Moore. The Use of a Formal Simulator to Verify a Simple Real Time Control Program. Technical Report ICSA-CMP-29, University of Texas at Austin, 1982.

14. F. Brown. An Investigation into the Goals of Research in Automatic Theorem Proving as Related to Mathematical Reasoning. Tech. Rept. 49, Department of Artificial Intelligence, University of Edinburgh, 1977.

15. R. Cartwright. A Practical Formal Semantic Definition and Verification System for Typed LISP. Tech. Rept. STAN-CS-77-592, Computer Science Department, Stanford University, 1976.

16. R. Cartwright. A Constructive Alternative to Axiomatic Data Type Definition. Conference Record of the 1980 LISP Conference, P.O. Box 487, Redwood Estates, Ca. 95044, 1980, pp. 46-55.

17. R. Cartwright and J. McCarthy. Recursive Programs as Functions in a First Order Theory. Tech. Rept. STAN-CS-79-717, Computer Science Department, Stanford University, 1979.

18. A.J. Cohn. "The Equivalence of Two Semantic Definitions: A Case Study in LCF." *SIAM Journal of Computing 12* (1983), 267-285.

19. A.J. Cohn and R. Milner. On Using Edinburgh LCF to Prove the Correctness of a Parsing Algorithm. Tech. Rept. CSR-112-82, University of Edinburgh, 1982.

20. R. L. Constable and M. J. O'Donnell. *A Programming Logic.* Winthrop, Cambridge, 1978.

21. J. Darlington. "An Experimental Program Transformation and Synthesis System." *Artif. Intell. 16,* 1 (1981).

22. M. Davis and J. Schwartz. Metamathematical Extensibility for Theorem Verifiers and Proof-checkers. Tech. Rept. 12, Courant Institute of Mathematical Sciences, 1977.

23. Benedetto Lorenzo Di Vito. Verification of Communications Protcols and Abstract Process Models. PhD Thesis ICSCA-CMP-25, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1982.

24. Richard J. Feiertag. A Technique for Proving Specifications are Multilevel Secure. Technical Report CSL-109, SRI International, 1981.

25. R. Floyd. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics,* American Mathematical Society, Providence, Rhode Island, 1967, pp. 19-32.

26. S. Gerhart. AFFIRM Type Library. USC Information Sciences Institute, 4676 Admiralty Way, Marina Del Rey, Ca. 90291, 1981.

27. J.A. Goguen. How to Prove Algebraic Inductive Hypotheses without Induction. In *5th Conference on Automated Deduction, Lecture Notes in Computer Science,* W. Bibel and R. Kowalski, Eds., Springer-Verlag, 1980, pp. 356-373.

28. D.I. Good, R.L. London, W.W. Bledsoe. An Interactive Program Verification System. Proceedings of 1975 International Conference on Reliable Software, 1975.

29. D. I. Good. The Proof of a Distributed System in Gypsy. Tech. Rept. ICSCA-CMP-30, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1982.

30. D. Good, et. al. Report on the Language GYPSY Version 2.0. Tech. Rept. ICSCA-CMP-10, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1978.

31. M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A Metalanguage for Interactive Proof in LCF. Tech. Rept. CSR-16-77, Department of Computer Science, University of Edinburgh, 1977.

32. P. Henderson. *Functional Programming.* Prentice-Hall International, Inc., London, 1980.

33. C. A. R. Hoare. "An Axiomatic Basis for Computer Programming." *Comm. ACM 12,* 10 (1969), 576-583.

34. C. A. R. Hoare. "Communicating Sequential Processes." *Comm. ACM 21,* 8 (1978), 666-677.

35. S. Igarashi, R.L. London, D.C. Luckham. Automatic Program Verification I: A Logical Basis and Its Implementation. Information Science Institute, USC, 1973. In Report ISI/RR-73-11

36. J. C. King. *A Program Verifier.* Ph.D. Th., Carnegie-Mellon University, 1969.

37. R. Kowalski. *Logic for Problem Solving.* Elsevier North Holland, Inc., New York, 1979.

38. L. Lamport. "Proving the Correctness of Multiprocess Programs." *IEEE Trans. Soft. Engrg. SE-3 2* (1977), 125-143.

39. C. Lengauer. On the Mechanical Transformation of Program Executions to Derive Concurrency. Technical Report TR-83-20, Department of Computer Sciences, University of Texas at Austin, 1983.

40. J. McCarthy. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hershberg, Eds., North-Holland Publishing Company, Amsterdam, The Netherlands, 1963.

41. J. McHugh. *Towards the Generation of Efficient Code from Verified Programs.* Ph.D. Th., University of Texas, 1984.

42. J S. Moore. "A Mechanical Proof of the Termination of Takeuchi's Function." *Information Processing Letters 9*, 4 (1979), 176-181.

43. D. Musser. On Proving Inductive Properties of Abstract Data Types. Proceedings of the Seventh ACM Symposium on Principles of Programming Languages, ACM SIGPLAN, 1980.

44. G. Nelson and D. C. Oppen. "Simplification by Cooperating Decision Procedures." *ACM Transactions of Programming Languages 1*, 2 (1979), 245-257.

45. S. Owicki and D. Gries. "Verifying Properties of Parallel Programs: An Axiomatic Approach." *Comm. ACM 19*, 5 (1976).

46. L. Paulson. Verifying the Unification Algorithm in LCF. Tech. Rept. Technical Report 50, University of Cambridge Computer Laboratory, 1984.

47. W. Polak. *Compiler Specification and Verification.* Springer-Verlag, Berlin, 1981.

48. M. Smith, A. Siebert, B. DiVitto, and D. Good. "A Verified Encrypted Packet Interface." *SIGSOFT 6*, 3 (1981).

49. D.F. Stanat, T.A. Thomas, and J.R. Dunham. Proceedings of a Formal Verification/Design Proof Peer Review. Tech. Rept. RTI/2094/13-01F, Research Triangle Institute, P.O. Box 12194, Research Triangle Park, N.C., 27709, 1984.

50. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Massachusetts, 1977.

51. D. A. Turner. "A New Implementation Technique for Applicative Languages." *Software -- Practice and Experience 9* (1979), 31-49.

52. United States of America Standards Institute. USA Standard FORTRAN. Tech. Rept. USAS X3.9-1966, United States of America Standards Institute, 10 East 40th Street, New York, New York 10016, 1966.

53. J. von Neumann. *John von Neumann, Collected Works, Volume V.* Pergamon Press, Oxford, 1961.

54. T.J. Wagner. *Hardware Verification.* Ph.D. Th., Stanford University, 1977.

55. R. W. Weyhrauch. "Prolegomena to a Theory of Mechanized Formal Reasoning." *Artificial Intelligence 13*, 1,2 (April 1980), 133-171.

56. R.W. Weyhrauch and A.J. Thomas. FOL: A Proof Checker for First Order Logic. Tech. Rept. AIM-235, Stanford University, Computer Science Department, Artificial Intelligence Laboratory, 1974.

END

FILMED

DTIC